VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

Vývoj testovacích modulů pro dohledový systém Nagios

Development of Testing Modules for the System Nagios

Radim Žurovec 2011

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 21. dubna 2011 . . . . . .

Tímto bych rád poděkovat Ing. Martinu Milatovi za cenné rady a připomínky při tvorbě této bakalářské práce.

Abstrakt

Cílem této bakalářská práce je sledování vybraných komponent distribuovaného systému Virtlab pomocí monitorovacího systému Nagios. Práce se z počátku zabývá analýzou tohoto monitorovacího systému a následně analýzou komponent virtuální laboratoře, dle souladu se zadáním bakalářské práce. Dále návrhem scénářů testování daných služeb a implementací testovacích skriptů pro každou s komponent distribuovaného systému, které jsou realizovány v souladu se zásadami tvorby modulů pro systém Nagios.

Klíčová slova: Virtlab, Nagios, Virtuální laboratoř, Centreon

Abstract

The aim of this bachelor‘s thesis is monitoring of selected components of a distributed system Virtlab using monitoring system Nagios. At the beginning the thesis analyzes this monitoring system and subsequent analysis the components of a virtual laboratory, according to submission for the thesis. Furthermore, the proposal scenarios of testing services and implementation of test scripts for each of the components of a distributed system that are implemented in accordance with the principles of realizing modules for system Nagios.

Keywords: Virtlab, Nagios, Virtuální laboratoř, Centreon

Seznam použitých zkratek a symbolů

TCP	- Transmission Control Protocol
IP	- Inernet Protocol
UDP	- User Datagram Protocol
CGI	- Common Gateway Interface
HTML	- Hypertext Markup Language
BASH	- Bourne Again Shell
OS	- Operation System
DNS	- Domain Name System
UML	- Unified Modeling Language
MVC	- Model View Controller
XML	- Extensible Markup Language
DTD	- Data Type Definition
ID	- Identificator
PHP	- Hypertext Preprocessor
Q-in-Q	- Tunelování 802.1q rámců v 802.1q rámcích
VLAN	- Virtual Local Area Network
ICMP	- Internet Control Message Protocol
ARP	- Address Resolution Protocol
ASSSK	- Automatizovaný Systém pro Správu Síťových Konfigurací

Obsah

Seznam obrázků

1 Úvod

Virtlab je projekt virtuální laboratoře počítačových sítí, jehož základní koncepce vznikla v roce 2005. Hlavním smyslem projektu je vzdáleně zpřístupnit studentům nákladné fyzické síťové prvky, zejména pro praktickou výuku studentů. Uživatelé si pak mohou vzdáleně přes internet rezervovat, libovolnou síťovou topologii a pracovat na úlohách skrze webové rozhraní z pohodlí domova.

Vzhledem k velkému množství komponent virtuální laboratoře občas dojde k výpadku některé z nich. Z důvodu dodržení nepřetržitého provozu Virtlabu, je nutno při výpadku co nejrychleji zasáhnout a laboratoř dostat do provozu schopného stavu. Z tohoto důvodu je třeba zavést do systému monitorovací systém, který správce uvědomí o výpadku některé z komponent virtuální laboratoře.

Tato bakalářská práce si klade za cíl analyzovat komponenty virtuální laboratoře, navrhnout a implementovat testovací aplikace pro systém Nagios, jež je v současné době úspěšně využíván pro monitorování serverů Virtlabu.

2 Dohledový systém Nagios

Systém Nagios je open source monitorovací nástroj, který je možno využít pro dohledávání infrastruktury (např. servery, směrovače, přepínače, tiskárny...) a služeb jimi poskytovaných. Do systému je možno přidat externí moduly pro sledování vlastních specifických síťových komponent.

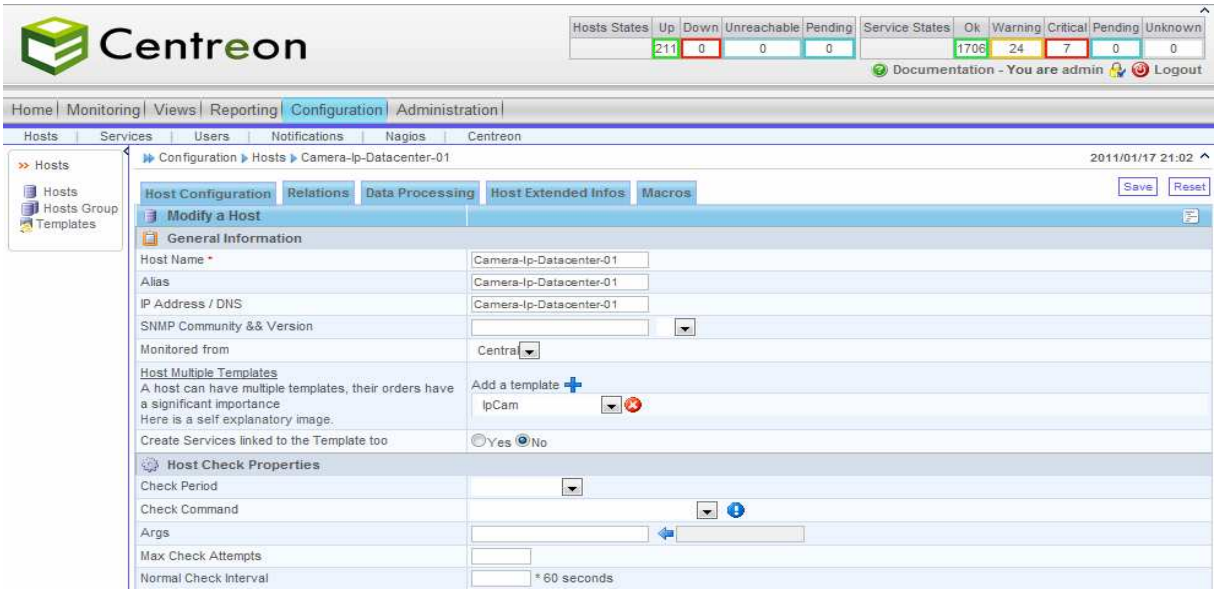
2.1 Architektura systému

Hlavní částí Nagiosu je démon nagios. Tento démon po spuštění načte z konfiguračních souborů nastavení a začne monitorovat. Informace o výsledcích testu ukládá do souboru či databáze. Z jiného souboru zase načítá příkazy. Pro zobrazení informací o stavu slouží webové rozhraní, které je realizováno několika CGI skripty. Tyto CGI skripty přistupují do souboru se stavy a zobrazují jej v rozumné formě jako HTML stránky. Pro přístup k Nagiosu můžeme použít HTML prohlížeč. Z webového rozhraní je možné do Nagiosu zasílat i jednodušší příkazy. Nevyhovuje-li koncovému uživateli z jakéhokoliv důvodu webové rozhraní, je možné webové rozhraní přizpůsobit (nacházíme se ve světě open source software) nebo napsat celé prezentační rozhraní znovu. Celý systém je navržen velmi logicky a pochopitelně[1].

2.2 Centreon – nadstavba systému Nagios

Hlavní nevýhodou systému Nagios je jeho čistě textová konfigurace v konfiguračních souborech, která je pro nezkušeného uživatele značně komplikovaná. Tento problém však odstraňuje nadstavba Centreon, která ve spolupráci z webovým serverem poskytuje jednoduchou konfiguraci skrze webové rozhraní. Centreon však nabízí spoustu jiných rozšíření, jako je podpora vykreslování grafů, přehlednější výpisy monitorovaných služeb apod. (Obrázek 1).

[1] Více informací o architektuře systému Nagios lze naleznout v [5] a [6]



Obrázek 1 - Ukázka grafického rozhraní systému Centreon

2.3 Zásady pro tvorbu modulů v systému Nagios

2.3.1 Systémové požadavky

Moduly dohledového systému Nagios jsou vyvíjeny typicky pro platformu GNU Linux, tedy modul psaný v libovolné technologii, kterou podporuje platforma GNU Linux může být spuštěn ze systému Nagios. V našem případě je tedy možno použít libovolnou technologii podporovanou systémem OS Linux. Tedy například: BASH, C/C++, Java, Perl, Python apod.

2.3.2 Výstupy z modulů

Pro získávání informací o průběhu testu, systém využívá standardního výpisu modulu (STDOUT). Z modulu můžeme vždy vytisknout zprávu o stavu testu, avšak měli bychom dbát na to, aby byl výstup co možná nejstručnější a nejvýstižnější. Tato zásada je nutná z důvodu možného zahlcení paměti logu. Jednořádkový výpis by tedy neměl být větší než 80 znaků. Služba zachytává zprávy pouze ze standardního výstupu, nikoli z chybového výstupu (STDERR). Není tedy možno použít chybový výstup. Jednu zprávu zásadně vypisujeme na jeden řádek. V modulu je možno využít tzv. upovídaný výstup (více řádkový výstup) a to spuštěním modulu s parametrem -v (verbose). Není však povinností modulu tento výpis podporovat.

Doporučený formát výstupu:

SYSTEM INFORMATION: some message
SYSTEM WARNNING: some message

2.3.3 Návratové kódy

Výpisy z modulu sice informují o stavu testu, avšak nelze z nich jednoznačně říci, jak test doopravdy v součtu aspektů dopadl (program skončil: v pořádku, chybou, varováním apod.). Pro tento jednoznačný aspekt služba využívá kladných návratových hodnot z programu.
Technologie ve které je modul vyvíjen tedy musí podporovat tyto návratové hodnoty. V běžných programovacích jazycích je tato skutečnost reprezentována metodou exit(int)
(java: *System.exit(int value)*, perl: *exit int*). Konkrétní návratové hodnoty a jejich význam jsou uvedeny v tabulce níže (Tabulka 1).

Návratový kód	Stav	Popis
0	V pořádku	Modul úspěšně ověřil stav služby a služba běží korektně
1	Varování	Modul úspěšně ověřil stav služby, avšak v průběhu testu byly překročeny některé prahové hodnoty (timeout apod.), jinak služba je schopná běhu
2	Chyba	Modul neověřil stav služby, v průběhu testu se vyskytly kritické chyby, služba není schopná běhu
3	Neznámý	Modul není schopen ověřit službu, příčina může být způsobena výpadkem sítě nebo interní chybou modulu (modul nebyl schopen načíst konfigurační soubor apod.), služba je tedy ve stavu neznámém

Tabulka 1 – Návratové kódy a jejich význam

2.3.4 Výstupní data

Někdy je potřeba z výstupu modulu vyčíst potřebná číselná data (např. pro vykreslení grafu).
Pro výstup dat je v modulech nagiosu určen speciální formát výstupu. Tento výstup se velice podobá klasické zprávě (může být i její součástí) a však data výstupu jsou oddělena znakem „|“.

Správný formát datového výstupu:

'label'=value[UOM];[warn];[crit];[min];[max]

- Label – název hodnoty datového výstupu (př. time, ratio), může obsahovat jakékoliv znaky
- Value – číselná hodnota výstupu, pokud UOM je v % nemusí být uvedeny min a max
- UOM – uvedená jednotka (B,kB,MB, %, ms apod.)
- Warn – varovná hodnota
- Crit – kritická hodnota
- Min – minimální hodnota
- Max – maximální hodnota

Přiklad výstupu:

INFO: test ok | 'time'=10ms;20;30;0;50

2.3.5 Rezervované volby

Ve skriptu je dovoleno použit jakýchkoliv rozšířených voleb. Výjimku však tvoří volby uvedené v seznamu, které nesmí být použity k ničemu jinému než je jejich skutečný význam.

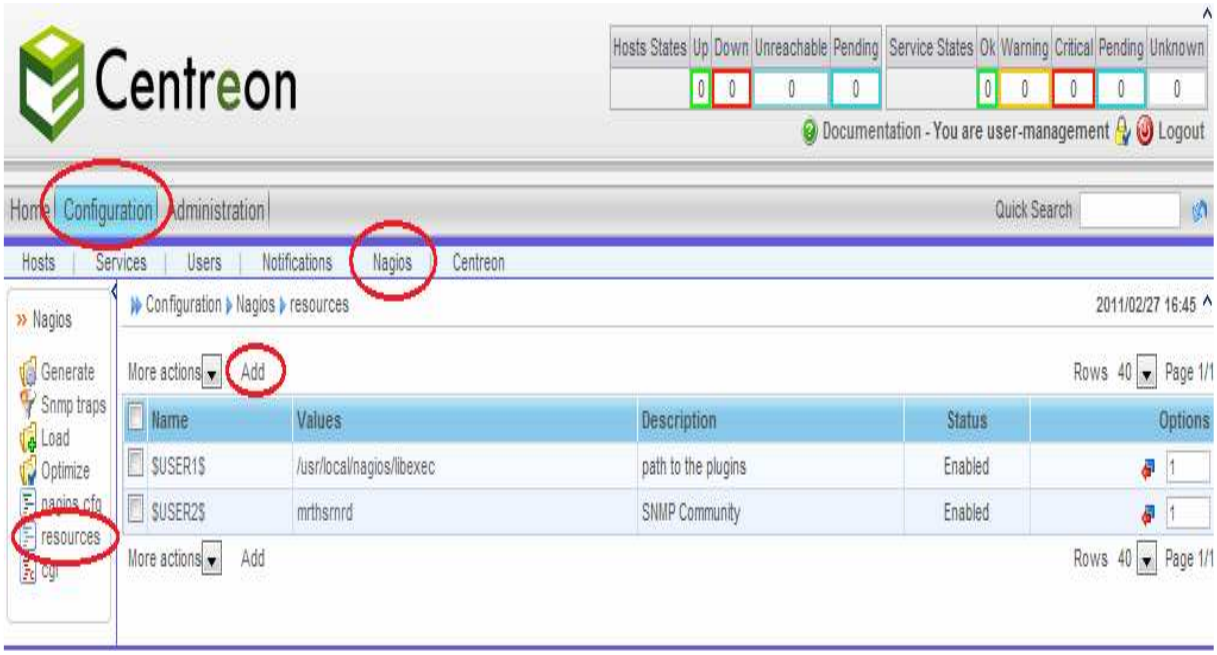
- -V (--version) – verze skriptu
- -v (--verbose) – upovídaný výstup
- -h (--help) – nápověda
- -t (--timeout) – maximální časová prodleva
- -H (--hostname) – název dotazovaného zařízení
- -w (--warnning) – varovná hodnota
- -c (--critical) – kritická hodnota

2.4 Zavedení modulu do systému

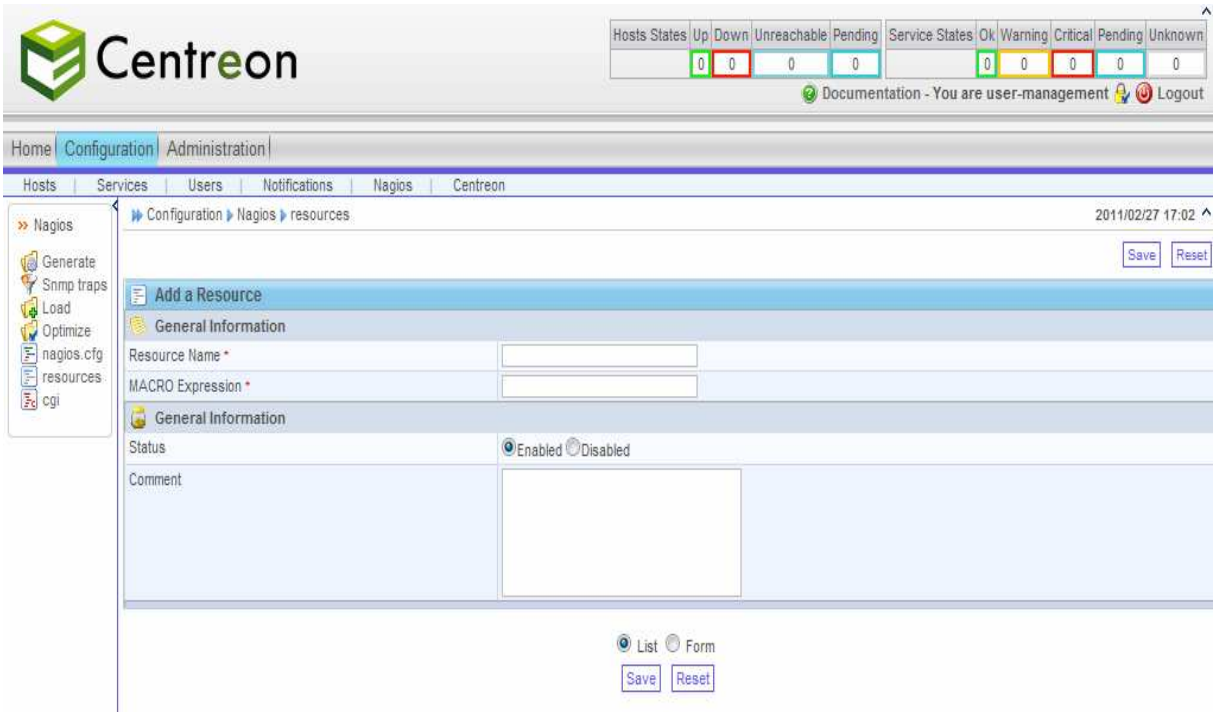
Aby systém Centron byl schopen monitorovat službu skrze námi napsaného skriptu, je třeba systému říct, kde skript najde, jakým způsobem a kdy ho má spouštět. Toto lze velice snadno nakonfigurovat v graficko-uživatelském rozhraní systému. Přesný postup konfigurace je uveden v podkapitolách níže.

2.4.1 Mapování kolekce modulů

Vezměme v úvahu příklad, že máme k dispozici kolekci jar souborů psaných v platformě Java a chceme jej zavést do systému Centreon. Nejprve musíme systému říci, kde se kolekce nachází. Tohle učiníme snadno definováním cesty k adresáři, kde se skripty nacházejí. Předpokládejme, že skripty máme uloženy v adresáři */usr/lib/java/nagios/*. Cesta ke skriptům se definuje přidáním nového zdroje (resource). Pomocí webového rozhraní přejdeme následující cestou: Configuration -> Nagios -> Resources a přes odkaz *add* se otevře menu pro definování nového zdroje (viz. Obrázek 2, Obrázek 3).



Obrázek 2- Centreon – konfigurace cesty



Obrázek 3- Centreon konfigurace cesty 2

Z obrázku je patrné, že systém po nás vyžaduje povinné položky *Resource Name* a *MACRO Expression*. Položka *Resource Name* reprezentuje název zdroje, na který se bude možno pod tímto názvem dotazovat. Název zdroje se standardně zapisuje velkými písmeny mezi znaky „$". Tedy např. *$JAVA1$.* Položka *MACRO Expression* reprezentuje cestu k nám napsaným skriptům, tedy /usr/lib/java/nagios/. Nastavení uložíme kliknutím na tlačítko „save".

2.4.2 Definování příkazu

V předchozí podkapitole jsme systému definovali umístění kolekce skriptů. Nyní musíme systému říci, jak se má skript spustit. K tomuto slouží konfigurační menu na které se dostaneme skrze webové rozhraní Configuration -> Commands -> Add. Toto menu má dvě povinné položky, a to *Command Name* a *Command Line*. Položkou *Command Name* definujeme název příkazu, který budeme spouštět jako celek pro danou službu. Položka *Command Line* vyjadřuje dotaz pro spuštění daného skriptu jako z příkazové řádky. Definovali jsme si, že máme kolekci jar souborů, vybereme tedy jeden ze nich a pojmenujeme jej například *test.jar*. Tento skript by se z příkazového řádku spouštěl následovně:

```
java –jar /usr/lib/java/nagios/test.jar
```

S využitím namapované cesty ke skriptům zápis zkrátíme na:

```
java –jar $JAVA1$test.jar
```

Rozšiřme náš příklad, a řekněme, že chceme skript spustit se dvěmi argumenty a to *–H,* který vyjadřuje IP adresu, nebo DNS název služby a *-p* který vyjadřuje port, na kterém služba naslouchá.

Argumenty v systému Centreon se definují řetězcem *$ARGN$*, kde *N* je pořadové číslo argumentu. Konkrétní hodnoty argumentů se dosazují při definici konkrétní služby na konkrétním stroji. *Command Line* našeho skriptu tedy bude mít výsledný tvar:

```
java –jar $JAVA1$test.jar –H $ARG1$ –p $ARG2$
```

Na konec nesmíme zapomenou skriptům nastavit oprávnění pro spouštění skriptů[2]:

```
chmod 111 /usr/lib/java/nagios/*.*
```

[2] Zajímavá připomínka z článku o systému Nagios z [8]

3 Monitorování komponent virtuální laboratoře

Následující kapitoly se zabývají komponentami virtuální laboratoře, jejich důkladnou analýzou a návrhem testování dané komponenty. U každé z nich uvedu návrhy scénářů testování a po té samotné implementační řešení. Všechny skripty jsou implementovány v jazyce Java, z důvodu kompatibility s operačním systémem Linux. Monitorované komponenty jsou následující: **Mazací server**, **Rezervační server**, **Konzolový server a Tunelovací server**.

4 Mazací server

Mazací server je důležitou komponentou ve virtuální laboratoři, kterou obsahuje každá z lokalit distribuovaného systému. Uživatelé, jež pracují na některém z fyzických prvků laboratoře (směrovač, přepínač apod.), mají možnost vzdáleně modifikovat nastavení zařízení, dokud uživateli nevyprší platnost jeho rezervace. Po skončení rezervace je vhodné, aby se zařízení uvedlo do nějakého předem určeného výchozího nastavení, jež dalšímu uživateli bude nejvíce vyhovovat. Tento problém má právě na starosti mazací server, na který je zaslán požadavek o smazání daného zařízení před začátkem nové rezervace uživatele. Mazací server je implementován v jazyce Python. Externí skripty v jazycích C a BASH.

4.1 Architektura a konfigurace

Jedná se o serverovou službu, která standardně naslouchá na TCP portu 60002. Mazací server je postaven na více vláknové technologii. Je tedy schopen obsloužit více klientů najednou (pro každého klienta se vytvoří jedno vlákno). Služba má k dispozici seznam zařízení, jež specifikují název zařízení, způsob jak se zařízením komunikovat a cestu k mazacímu skriptu, který není nic jiného, než seznam příkazů, které uvedou zařízení do výchozího zvoleného stavu. Seznam zařízení může být pro každou lokalitu rozdílný, proto je tento seznam zapsán v textovém konfiguračním souboru mazacího serveru *erase-device.conf*. Zařízení může být dvojího typu. První typ je osobní počítač, kdy server je schopen s počítačem komunikovat prostřednictvím TCP protokolu. Je tedy nutné předem znát IP adresu a port na kterém počítač naslouchá . Druhým typem je zařízení ku příkladu Cisco box, který je s lokalitou propojený pomocí sériové linky. S takovýmto typem zařízení je možno komunikovat pouze prostřednictvím emulovaného terminálu nad sériovou linkou VTY 100.

Struktura textového konfiguračního souboru *erase-device.conf*, je zcela jednoduchá. Pro každé zařízení je vyhrazen jeden řádek oddělen znakem <LF>, kde se nachází v první části název zařízení. V první části argumentů se nachází název zařízení, jež zprostředkuje upload skriptu. Dále cesta k ovladači pokud se jedná o zařízení CISCO, nebo IP adresa a port ve tvaru IPADRESA:PORT jedná-li se o počítač. Dále cesta k mazacímu skriptu a prodleva mezi zaslanými znaky v milisekundách. Všechny argumenty se oddělují jednou mezerou.

Konfigurační soubor pro dvě zařízení může mít následující tvar:

```
r1@ostrava /opt/virtlab/erase-server/utils/upload-conndevice-cfg
/dev/ttyM0 /opt/virtlab/erase-server/erasers/erase-cisco 0
pc1@ostrava /opt/virtlab/erase-server/utils/upload-conndevice-cfg
10.0.0.20:5555 /opt/virtlab/erase-server/erasers/erase-pc 0
```

Dostupnost Mazacího serveru ověříme příkazem:

```
telnet localhost 60002
```

4.2 Komunikační protokol:

Komunikační protokol mazacího serveru je textově orientovaný protokol, který je realizován prostřednictvím TCP protokolu. Po navázání TCP spojení se službou, se uživatel ocitne v jakési virtuální konzoli, kdy má k dispozici celkem čtyři volby (uvedeny níže). Každá volba se odděluje znakem ukončení řádku (<LF>).

- **exit** – slouží ke korektnímu odpojení od služby
- **help** – vypíše nápovědu na konzoli
- **show devices** – vypíše seznam zařízení které lze smazat. K zařízením uvede i jejich ovladač
- **erase <*název zařízení*>** – požadavek o smazání zařízení, tento příkaz má jeden argument a to název zařízení, které požadujeme smazat

4.3 Soubor akcí pro smazání zařízení:

4.3.1 Strana klienta

Klient se pomocí TCP protokolu připojí na mazací server a zašle požadavek o vypsání všech zařízení (show devices). Pokud seznam obsahuje zařízení, které klient požaduje vymazat, pomocí příkazu *erase <název zařízení>* zařízení smaže. Vzhledem k tomu, že mazací server žádným způsobem neodpovídá na žádost o smazání fyzického prvku, je prakticky nemožné zjistit, zda zařízení bylo opravdu vráceno do výchozí konfigurace.

4.3.2 Strana serveru

Server přijme od klienta TCP spojení a pro souběžnou komunikaci vytvoří nové vlákno. Po požadavku na vypsání všech prvků zašle pomocí TCP linky jejich seznam. Po zaslání požadavku na smazání zařízení server ověří, zda název prvku odpovídá některému názvu z tohoto výpisu. Pokud ověření proběhne v pořádku, služba dohledá dodatečné informace o zařízení a volá externí program pro upload mazacího skriptu. Tímto práce mazacího serveru končí. Z tohoto je patrné, že mazací server nemůže ověřit, zda se mazací skript vůbec dostal k danému zařízení.

4.4 Návrh scénáře testování mazacího serveru

Z důvodu řádného testování funkčnosti mazacího serveru jsem se rozhodl, že vytvořím fiktivní zařízení, reprezentované vzdáleným počítačem, které uvedu v seznamu fyzických prvků a pojmenuji jej např. *test@device*. Po připojení a odeslání žádosti o smazání tohoto zařízení se skrze tento záznam server připojí na podstrčenou IP adresu, která bude vést datový tok zpět na testovací aplikaci a odešle mazací skript v domluveném tvaru (např. Hello world!). Aplikace tedy má za úkol porovnat výsledek přijatých dat s předpokládaným, a pokud výsledky souhlasí, funkčnost služby je v pořádku.

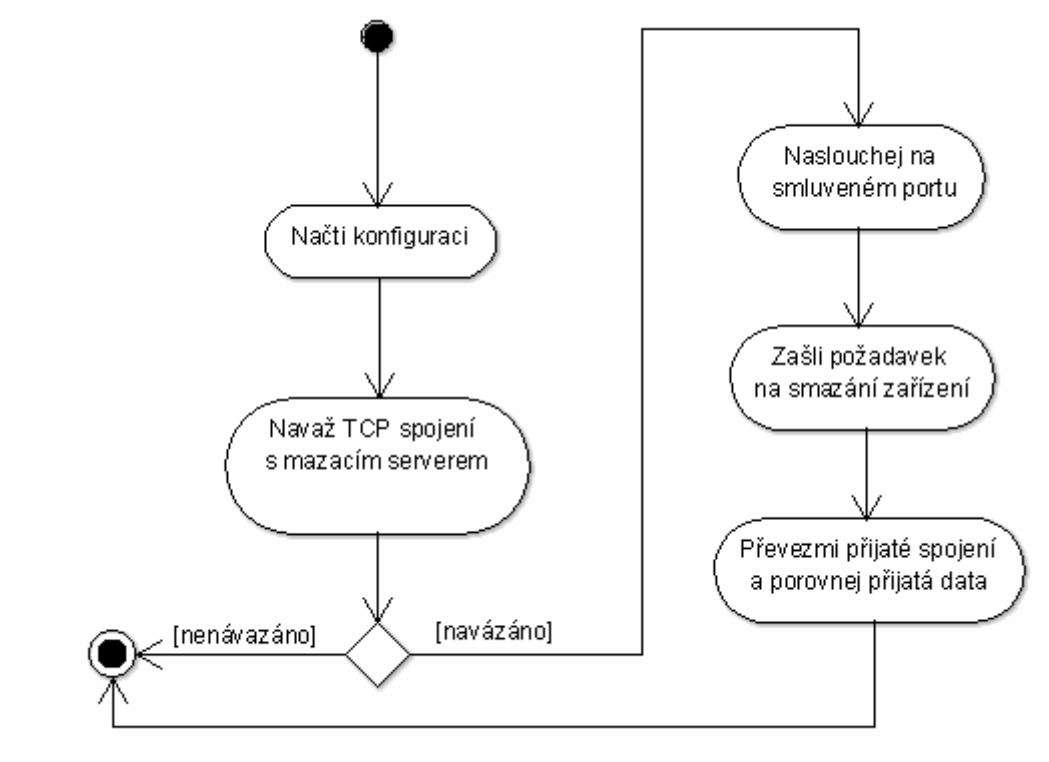
Záznam o fiktivním zařízení by mohl mít tvar:

```
test@device /opt/virtlab/erase-server/utils/upload-conndevice-cfg
156.125.115.3:5550 /opt/virtlab/erase-server/erasers/testovaci-
skript 0
```

4.5 Návrh a implementace testovacího skriptu

4.5.1 Diagram aktivit

Základní tok událostí, které by se měly odehrát v testovacím modulu, jsou zobrazeny v zjednodušeném UML aktivitním diagramu níže (Obrázek 1).



Obrázek 4 – Scénář testování mazacího serveru

Aktivita Načti konfiguraci vyjadřuje načtení základních informací pro běh aplikace. Tyto informace obsahují údaje o testované službě (umístění v síti, příkaz ke smazání zařízení), dále také údaje o běhu testu (timeout, číslo portu serverového socketu). Po načtení konfigurace se aplikace pokouší o

navázání spojení s mazacím serverem. Pokud se spojení nepodaří navázat, běh aplikace končí. V opačném případě přecházíme k další aktivitě Naslouchej na smluveném portu. Tato aktivita musí být vykonána před odesláním požadavku o smazání zařízení, protože inicializace serverového socketu je relativně časově drahá operace. Mohlo by se tedy stát, že po odeslání zprávy by se mazací server pokoušel navázat spojení ještě dříve, než by byl inicializován serverový socket a nezaslal by potřebná data ke srovnání, což by vedlo k chybnému vyhodnocení funkčnosti služby.

4.5.2 Rozdělení do komponent

Z hlediska možného rozšíření jsem se rozhodl aplikaci rozdělit do tří nezávislých komponent dle návrhové vzoru MVC.

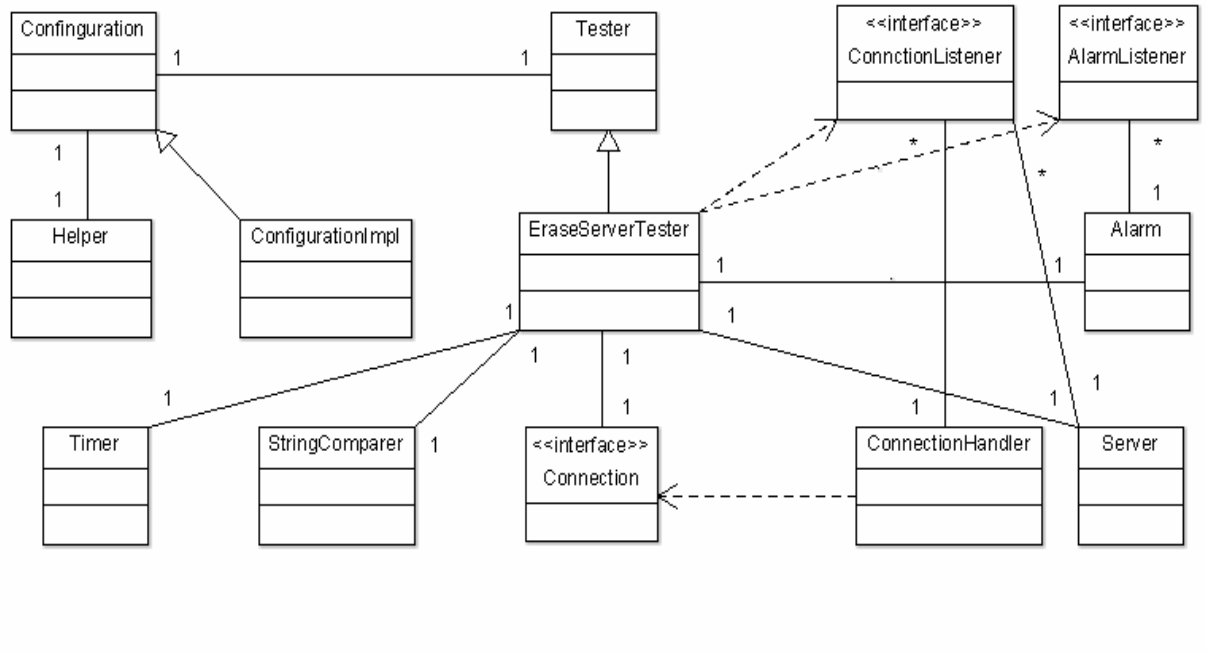
Konfigurace – Komponenta má za úkol získat potřebné informace k realizaci testu. Tedy zpracování dat z konfiguračního souboru a argumentů při spouštění aplikace. Jako konfigurační soubor jsem zvolil XML dokument, z důvodu jeho přehlednosti a snadného ověřování platnosti. Je jedinou komponentou, pomoci které může uživatel ovlivnit průběh testu.

Řízení testu – Komponenta se stará o řízení celého průběhu testu. Určuje tedy, kdy se má načíst konfigurace, navázat spojení se službou, naslouchat pro příchozí spojení, odesílat a přijímat data. Dále se také stará o zachycení a správné reakci na možnost vyvolaných výjimek, správné formátování výstupních dat, či vyhodnocení testu.

Komunikace – Komponenta reprezentuje spojení mezi aplikací a serverem. Určuje, jakým způsobem se budou data odesílat, či přijímat.

4.5.3 Návrh tříd

Strukturu aplikace vyjadřuje UML třídní diagram na obrázku (Obrázek 5).



Obrázek 5 – Architektura testovací aplikace mazacího serveru

4.5.4 Význam důležitých tříd a rozhraní

Třída : Configuration

Jedná se o abstraktní třídu, jejichž instance definuje všechny konfigurační vlastnosti celé aplikace. Zděděná třída má za úkol libovolným způsobem načíst tyto vlastnosti a poskytnout řídící třídě, jež tyto vlastnosti použije. Třída také poskytuje svou implementaci a to vnitřní třídou ConfigurationImpl. Tato implementace je dostupná pomoci metody *static Configuration getConfiguration(Helper h).*

Třída: ConfigurationImpl

Jedná se o vnořenou (vnitřní) třídu, jejichž instance dědí z třídy Configuration. Pro načtení konfiguračních hodnot jsem použil XML konfigurační soubor s názvem *conf.xml*. Který musí být umístěný v adresáři aplikace. Pro načítání souboru jsem použil knihovnu org.w3c.parsers [3].

Třída : Tester

Tato třída abstraktně definuje objekt, který odpovídá specifikaci testovacího modulu systému Nagios. Instance této třídy má za úkol řídit průběh a zpracování celého testu.

Třída : EraseServerTester

Jedná se o konkrétní implementaci abstraktní třídy Tester, která má za úkol řídit průběh testu, načítání konfigurace, odchytávání chybových hlášení, formátování výstupních dat apod.

Třída : Alarm

Tato třída má za úkol upozornit řídící třídu (EraseServerTester) o vypršení maximální prodlevy aplikace. Pro realizaci funkčnosti a znovu použitelnosti jsem využil návrhového vzoru Listener kdy, po vypršení stanovené doby třída upozorní všechny posluchače o této události. Jelikož se jedná o paralelní činnost, třída je spouštěna v samostatném vlákně (dědí z třídy Thread).

Třída : ConnectionHandler

Jedná se o třídu, která zajišťuje TCP spojení s mazacím serverem pomocí třídy java.net.Socket. Slouží zejména pro odesílání dat směrem k mazacímu serveru.

Třída : ServerConnection

Jde o třídu, která zajišťuje TCP naslouchání na daném portu, pro příchozí spojení z mazacího serveru. Pro přijímáni dat je vytvořeno nové vlákno, které po ukončení přenosu informuje řídící třídu o ukončení přenosu pomoci návrhového vzoru Listener.

[3] Informace o parsování XML dokumentů v jazyce Java lze najít v [1]

5 Rezervační server

Jeden z požadavků na funkčnost virtuální laboratoře je sofistikovaný rezervační systém, kdy si uživatelé mohou ve zvoleném čase rezervovat libovolnou úlohu. Tento systém řídí komponenta s názvem Rezervační server, která poskytuje řídícímu (webovému) serveru informace o volných zařízeních, ale také umožňuje uložení, či zrušení záznamů o rezervacích. Služba je implementována v jazyce C++ a spolupracuje s databází MySQL (viz. odkaz [4]).

5.1 Architektura a konfigurace

Jedná se o serverovou službu typu požadavek – odpověď (request – response), která standardně naslouchá na TCP portu 50001. Vzhledem k paralelnímu zpracování požadavků, systém využívá více vláknový přístup (pro každý požadavek se vytvoří vlastní vlákno). Služba má k dispozici seznam základních časových plánů, ve kterém může realizovat rezervace a který se načítá při spouštění aplikace. Tyto časové plány se konfigurují v souboru *rsv-server.conf*, který obsahuje nejen časové plány ale i údaje o lokalitě a přístupu do databáze. Jelikož je virtuální laboratoř distribuovaný systém, služba poskytuje rezervovat síťové prvky z jiných lokalit, je tedy třeba definovat i časové plány vzdálených síťových prvků. Vzdálené lokality se definují řádkem, který začíná klíčovým slovem *virtlab*. Za klíčovým slůvkem následuje mezera, název a IP adresa lokality. Pod tento řádek se již definují názvy zařízení a přidělení do časové tabulky odpovídající časovému rozvrhu dané lokality. Časový plán se dá nakonfigurovat i za běhu aplikace a to v přístupové konzoli, která běží paralelně se službou a naslouchá na TCP portu 50002. Dále služba disponuje informacemi o samotných zařízeních dané lokality. Tyto informace jsou obsaženy v XML dokumentu *vybaveni.xml* s DTD restrikcí v souboru *equipment.dtd*.

Struktura časového plánu je velice snadná. Jde o textově zapsané hodnoty jednotlivých dní (sunday, monday, tuesday apod.) a ke každému dni přiřazeny přiřazen rozsah hodin oddělen jednou mezerou. Rozsah hodin se udává ve tvaru HH-HH a každý den se odděluje znakem ukončení řádku (<LF>).
Časový plán pro celý týden a jedno vzdálené zařízení může mít následující tvar.

```
timetable tt1
sunday 0-24
monday 10-12
monday 16-18
tuesday 9-18
wednesday 9-16
thursday 10-12
friday 0-24
saturday 0-24
virtlab karvina-virtlab 192.168.87.88
r1@karvina tt1
```

Struktura souboru *vybaveni.xml* je vskutku komplexní. Jedná se o velice proprietální XML dokument, který obsahuje velice detailní informace o zařízeních.[4]

[4] Detailnější informace o architektuře rezervačního serveru naleznete v Diplomové práci Radka Nováka [2]

5.2 Komunikační protokol

Jde o textově orientovaný protokol, využívající TCP spojení. Každý požadavek začíná příkazem, který se odděluje znakem <LF>. Po té následují argumenty, které se rovněž oddělují znakem ukončení řádku a jejich hodnoty znakem „:“ (výjimku tvoří argument *time*, kde je zápis složitější). Požadavek se ukončuje dvěma znaky <LF>. Aktuální typy požadavku jsou následující:

- **GET-OFFER** – vrátí nabídku zařízení dané lokality, nebo celého distribuovaného systému
- **RESERVE** – rezervuje dané zařízení na daný rozsah v čase
- **CANCEL** – zruší danou rezervaci
- **COMMIT** – potvrdí nepotvrzenou rezervaci
- **ATTACH** – uloží konfiguraci topologie pro danou rezervaci (tímto požadavkem se nebudeme zabývat)

5.2.1 Požadavek GET-OFFER

Jedná se o požadavek zjištění nabídky zařízení dané lokality, či celého distribuovaného systému. Tento požadavek lze ještě rozdělit na pod kategorie, kdy budeme po serveru chtít zjištění obecné nabídky (nezávislé na čase), a konkrétní nabídky (na dané časové rozmezí). Požadavek na zjištění obecné nabídky může mít následující tvar.

```
GET-OFFER
FOR:muj-virtlab
USER-GROUP:mojeskupina
```

Požadavek pro zjištění konkrétní nabídky celého distribuovaného systému

```
GET-OFFER
FOR:muj-virtlab
TIME:from{2011-01-01 10:35}to{2011-01-01 12:20}
USER-GROUP:mojeskupina
DISTRIB:yes
```

Význam jednotlivých argumentů:

- **FOR**- jako hodnotu předáváme název lokality rezervačního serveru, povinný atribut
- **USER-GROUP** – jedná se o název skupiny, ke které uživatel patří, povinný atribut
- **TIME** – jde o čas od/do ve formátu from{YYYY-MM-DD HH-MM}to{YYYY-MM-DD HH-MM}, nepovinný atribut
- **DISTRIB** – nabývá hodnot yes/no pokud chceme nabídku celého distribuovaného systému, či nikoliv, nepovinný atribut (implicitně nabývá hodnoty „no“)

Jako odpověď služba vrací:

- Číselný kód (200,300)
- Velikost zaslaných dat ve znacích
- Seznam zařízení v XML datech

5.2.2 Požadavek RESERVE:

Jde o požadavek o uložení rezervace, pro danou uživatelskou skupinu. Tento požadavek lze rozdělit na dvě podskupiny, a to požadavek s potvrzením a požadavek bez potvrzení. Pokud zašleme na server požadavek bez potvrzení, je nutno pro zohlednění rezervace později zaslat požadavek COMMIT.
Příklad požadavku RESERVE s potvrzením:

```
RESERVE
ID:1@virtlab
FOR:virtlab-virtual
USER-GROUP:mojeskupina
TIME:from{2011-01-01 10:35}to{2011-01-01 12:20}
DEVICE:mydevice@virtlab
```

Příklad požadavku RESERVE bez potvrzení:

```
RESERVE
ID:1@virtlab
FOR:virtlab-virtual
USER-GROUP:mojeskupina
TIME:from{2011-01-01 10:35}to{2011-01-01 12:20}
NEED-COMMIT:no
DEVICE:mydevice@virtlab
```

Význam argumentů:

- **ID** – jedná se o jednoznačný identifikátor dané rezervace ve formátu číslo@lokalita
- **FOR** – jako hodnotu předáváme název lokality rezervačního serveru, povinný atribut
- **USER-GROUP** – jedná se o název skupiny, ke které rezervace patří, povinný atribut
- **TIME** – jde o časové rozmezí od/do ve formátu {YYYY-MM-DD HH-MM}to{YYYY-MM-DD HH-MM}, povinný atribut
- **NEED-COMMIT** – nabývá hodnot yes/no jestliže chceme rezervaci předem potvrdit, či nikoliv, nepovinný atribut (implicitně „yes“)
- **DEVICE** – reprezentuje název zařízení, jež chceme rezervovat, jelikož v jednom požadavku můžeme rezervovat více zařízení, může se v požadavku atributů DEVICE vyskytovat více, než jeden

Služba vrací odpověď ve tvaru návratového kódu (200 RESERVED, 300 NOT-RESERVED).

5.2.3 Požadavek COMMIT

Požadavek COMMIT slouží k potvrzení zatím nepotvrzené rezervace. Pokud uživatel zašle požadavek s hodnotou atributu NEED-COMMIT „no“, je nutno pro potvrzení zaslat právě požadavek COMMIT. Jinak bude po určitém čase rezervace uvolněna s paměti rezervačního serveru. Požadavek obsahuje pouze jeden argument (ID), který obsahuje jednoznačný identifikátor předchozí rezervace.

Příklad požadavku COMMIT:

```
COMMIT
ID:1@virtlab
```

Odpověď na požadavek je ve tvaru *200 COMMITED*, nebo *300 NOT-COMMITED*.

5.2.4 Požadavek CANCEL

Požadavek slouží k zrušení rezervace. Dotaz má pouze jeden argument, a to jednoznačný identifikátor rezervace (ID), kterou požadujeme zrušit.

Příklad požadavku CANCEL:

```
CANCEL
ID:1@virtlab
```

Odpověď na požadavek je ve tvaru *200 CANCELED*, nebo *300 NOT- CANCELED*.

5.3 Návrh scénáře testování rezervačního serveru

5.3.1 Scénář č.1

Tento scénář spočívá v jednoduché myšlence, a to zaslat pouze požadavek GET-OFFER, který zjistí zda-li jsou dostupná nějaká zařízení. Jestli jsou, nebo nejsou zařízení volná není podstatné, protože v reálném provozu se často stává, že jsou všechna zařízení rezervována. Úkolem testu tedy je ověřit správnost návratového kódu, který nám služba vrátí jako odpověď požadavku GET-OFFER. Testem se ověří především dostupnost služby. Výhodou testu je především jeho jednoduchost a to z důvodu nezatěžování služby samotné (zvláště při větší periodičnosti ověřování).

5.3.2 Scénář č.2

Úkolem scénáře je opět zaslat požadavek GET-OFFER, který vrátí seznam volných zařízení. Vybereme jedno z dostupných zařízení a následně jej rezervujeme. Pokud není k dispozici žádný ze síťových prvků, vytvoříme prázdnou rezervaci. Po úspěšném rezervování rezervaci zrušíme požadavkem CANCEL. Funkčnost služby ověříme opět pomocí návratových kódu daných požadavků. Testem se ověří dostupnost služby a funkčnost databáze.

5.3.3 Scénář č.3

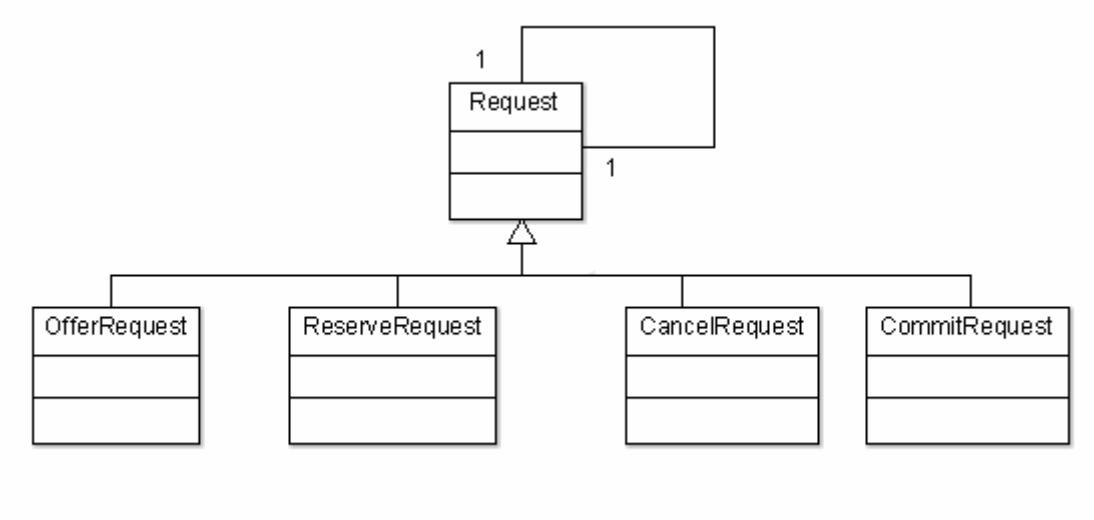
Tento scénář je velice podobný scénáři č.2 (5.3.2), a však můžeme předpokládat, že služba nezasílá vždy správný návratový kód. Tedy zašleme požadavek GET-OFFER, který zjistí zda-li jsou volné prvky. Opět vybereme jeden prvek, který následně rezervujeme. V průběhu ověřujeme návratové kódy. Pokud je prvek rezervován a služba pracuje správně, v nabídce pro daný čas by se tedy zařízení nemělo vyskytovat. Proto zašleme opět požadavek GET-OFFER a z výsledku zjistíme, zda-li

v nabídce prvek není. Následně rezervaci zrušíme požadavkem CANCEL. Test podrobně ověří dostupnost služby a databáze. Nevýhodou však je jeho časová náročnost na vykonání, zejména ve provozní špičce by mohl test omezovat datovou propustnost služby.

5.4 Návrh a implementace testovacího skriptu

Vzhledem k široké rozmanitosti scénářů, jsem se rozhodl, že implementuji skript univerzálnějšího ražení, kdy se uživatel aplikace může sám rozhodnout, kdy a jaké požadavky na službu pošle a podle jakých kritérií se bude rozhodovat o úspěšném vykonání testu.

5.4.1 Objektový model požadavků



Obrázek 6 – Objektový model požadavků

Třída Request je abstraktní třída, která zapouzdřuje konkrétní typy požadavků. Je také využita z důvodu možného rozšíření aplikace, či změny komunikačního protokolu na straně rezervačního serveru. Třída má vazbu sama na sebe, a to z důvodu možné komunikace mezi samotnými požadavky. Např. při rezervaci potřebuji znát název některého zařízení z předchozího požadavku na nabídku. Kdyby nebyly požadavky propojeny, nemohli bychom tuto hodnotu žádný způsobem zjistit. Instance tříd se vytvářejí najednou, a to při načtení konfigurace aplikace.

5.4.2 Implementace konfigurace

Z důvodu nastavování scénáře testu, jsem se rozhodl vytvořit speciální konfigurační soubor. Jako typ souboru jsem navrhl XML dokument, a to kvůli jeho stromové struktuře, přehlednosti a jednoduché manipulaci.

Klíčem k definici požadavku slouží značka request, která má dva povinné atributy *id* a *type*. Id vyznačuje pořadové číslo požadavku, a type reprezentuje daný typ požadavku (offer, reserve, commit, cancel).

Příklad definice požadavku:

```
<request id="1" type="offer">


</request>
```

Jelikož budeme chtít přidat také nějaké argumenty do požadavku, navrhl jsem vnořené značky, které budou reprezentovat dané argumenty. Značka bude mít název *attribute* a vlastnit jeden atribut *name*, který vyznačuje název argumentu (id, time, for, user-group apod.). Mezi značkami attribute, se již nachází konkrétní hodnota atributu. Některé atributy není potřeba implicitně definovat. Zejména ty, které se zadávají z příkazové řádky. Např. za argument *for* se dosadí parametr lokality z příkazové řádky.

Hodnota některých argumentů nemusí být předem známá. Například před zasláním nabídky zařízení předem nevíme, které zařízení je volné, a tedy nemůžeme jednoznačně specifikovat síťový prvek, který se má rezervovat. K vyřešení tohoto problému jsem navrhnul značku, jež bude reprezentovat vazbu na jiný požadavek a bude vracet název některého parametru z odpovědi daného požadavku. Značku jsem nazval *dependecy-on* a vlastní dva povinné atributy *request* a *property*. Atribut *request* identifikuje požadavek, na něž chceme aplikovat závislost. Jako hodnotu předáme jeho pořadové číslo (id). Atribut *property* vyznačuje název parametru z odpovědi požadavku (u požadavku nabídky je název *device*).

Příklad aplikace vazby na jiný požadavek

```
<request type="offer" id="1">

</request>
<request type="reserve" id="2">
     <attribute name="device">
          <dependency-on request="1" property="device" />
     </attribute>
</request>
```

Pro definování omezení, nebo-li co se má ověřovat, jsem navrhnul vnořenou značku *restrictions.* Značka je bez argumentů a musí být obsažená v každé definici požadavku. Mezi značky *restrictions* se vkládají konkrétní omezovací typy. Typy jsou implementovány celkem dva, a to ověření pomocí návratové hodnoty a ověření porovnáním. Ověření pomocí návratové hodnoty se definuje značkou *response-code*. Značka má jeden atribut *code*, který vyjadřuje předpokládaný návratový kód odpovědi (z pravidla bývá 200). Ověření porovnáním reprezentuje porovnání, zda-li se v odpovědi nachází daná hodnota, či ne. Ověření porovnání reprezentuje značka *equal*, která má tři povinné atributy *request*, *property* a *mode*. Atribut *request* vyjadřuje jednoznačný identifikátor požadavku, se kterým chceme porovnávat. Atribut *property* značí název parametru požadavku (většinou device) a mód vyjadřuje, zda-li má vyhodnotit omezení kladně, či záporně. Nabývá tedy hodnot *negative* nebo *positive*.

Příklad zápisu omezení:

```
<restrictions>
     <response-code code="200" />
     <equal request="1" property="device" mode="negative" />
</restrictions>
```

Důležitým aspektem v rezervačním serveru je čas. Pokud uživatel bude požadovat, aby se aplikace přizpůsobila časovým změnám, bylo by vhodné implementovat podporu práce s aktuálním časem. Tedy na příklad uživatel požaduje vybrat aktuální nabídku zařízení a v aktuálním čase rezervovat zařízení. Proto tyto případy jsem implementoval podporu výrazů, které vrací čas ve vhodném formátu (YYYY-MM-DD HH:MM). Výraz se zapisuje do složených závorek a výchozí hodnota je číslo 0, které odpovídá datu 1.1. 1970 00:00. Tedy každá kladná hodnota je brána jako počet milisekund od výchozího data (1.1. 1970 00:00). Vzhledem k přizpůsobení se aktuálnímu času, je možno využít proměnné *#SYSTIME* , která vrací aktuální časovou hodnotu. Výrazy podporují pouze operátor „+“, je tedy možné aktuální čas posunout pouze směrem do budoucna.

Příklad zápisu časově dynamické rezervace (od – aktuální čas, do – aktuální čas + 1 minuta):

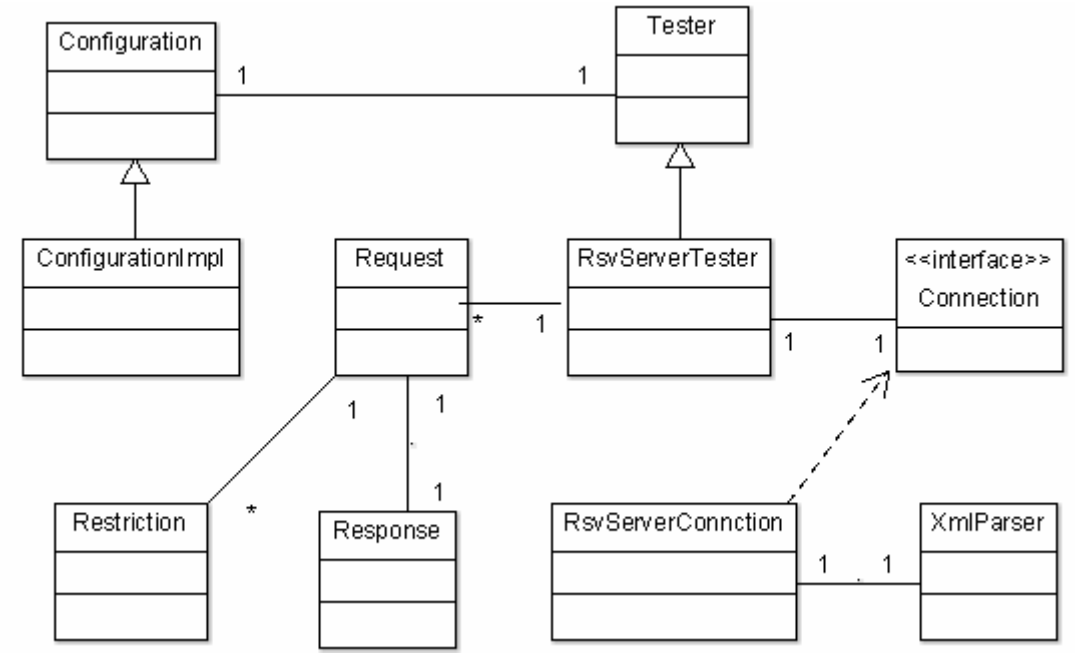
```
<request id="2" type="reserve">
     <attribute name="from">{#SYSTIME}</attribute>
     <attribute name="to">{#SYSTIME + 60000}</attribute>

</request>
```

Vzhledem k náročnosti na konfiguraci požadavků jsem připravil již předpracované konfigurační soubory, které odpovídají scénářům uvedených v předchozí kapitole 5.3. Je tedy na uživateli, zda použije jeden z nich, nebo si nakonfiguruje vlastní[5].

[5] Detailnější informace o konfiguraci naleznete v souboru ReadMe.txt, který je přiložen spolu se samotným skriptem

5.4.3 Architektura aplikace

Architekturu aplikace znázorňuje zjednodušený UML třídní diagram níže (Obrázek 7).



Obrázek 7 – Objektový model testovací aplikace pro rezervační server

5.4.4 Význam důležitých tříd

Třída Configuration:

Jedná se o abstraktní třídu, která definuje objekt poskytující informace o celkovém nastavení aplikace a také seznam požadavků, které se mají na server zaslat. Třída poskytuje svou implementaci, a to třídou ConfigurationImpl, která je dostupná pomocí metody *static Configuration getConfiguration(Helper h)*

Třída ConfigurationImpl:

Jde o konkrétní implementaci konfigurace. Jako konfigurační soubor, jsem zvolil XML dokumenty, kdy jeden poskytuje základní informace o službě (port, IP Adresa apod.), a druhý určuje scénář zasílání požadavků.

Třída Request:

Tato třída abstraktně definuje objekt požadavku. Zděděná třída má především za úkol implementovat metodu *String stringRepresentation(),* která vrátí textovou reprezentaci požadavku ve správném formátu. Obsahuje také seznam restrikcí, definovaných uživatelem, které se aplikují po zaslání požadavku a vazby na jiné požadavky.

Třída Response:

Třída Response je abstraktní třída, jejichž instance má na starosti zpracovat výstupní data ze zaslaného požadavku, které se zadávají jako parametr při vytváření konkrétní instance. Třída spolupracuje s třídou *XmlParser*, která jí překládá vrácená XML data do rozumnější podoby. Její konkrétní implementace jsou definovány třídami *OfferResponse*, *ReserveResponse*, *CommitResponse* a *CancelResponse*.

Rozhraní Restriction:

Jedná se o rozhraní, které definuje jedinou metodu, a to metodu *boolean check(List<Request> req).* Pomocí této metody řídící třída vyhodnocuje správnost odpovědi z rezervačního serveru. Rozhraní implementují dvě třídy *ParameterEqualRestriction* a *ResponseCodeRestriction* reprezentující omezení popsány v předchozí kapitole.

Třída RsvServerConnection:

Třída zajišťuje síťové spojení mezi aplikací a rezervačním serverem. Jejími úkoly jsou, vytvořit komunikační linku pomocí TCP protokolu, přes vytvořené spojení zaslat požadavek a přijmout odpověď z rezervačního serveru. Třída implementuje rozhraní *Connection*, jež definuje obecné síťové spojení.

Třída RsvServerTester:

Jedná se o třídu, která řídí celý průběh testu. Zajišťuje správné formátování výstupních dat, zasílání požadavků, kontrolu odpovědí, zachycení a správnou reakci na možný výskyt výjimek, načítání konfigurace atd.

6 Konzolový server

Uživatelé, jež chtějí pracovat na některém ze vzdálených prvků virtuální laboratoře požadují, aby mohli pro snadnou manipulaci se zařízeními přistupovat pomoci jakési virtuální konzole. Úkolem konzolového serveru tedy je umožnit uživateli, jež nemá přímý přístup k laboratorním prvkům, zprostředkovat vzdálené propojení mezi konzolí a konkrétním zařízením. Konzole je realizována Java Appletem a běží na webovém rozhraní. Konzolový server je implementován v jazyce C++ a konzole v jazyce Java [4].

6.1 Architektura a konfigurace

Jde o serverovou službu, která standardně naslouchá na TCP portu 10000. Vzhledem k pokrytí více paralelně pracujících uživatelů, je služba postavená na více vláknové technologii, tedy pro každé propojení (uživatel – zařízení) se pro obsluhu vytvoří samostatné vlákno. Konzolový server obsahuje seznam zařízení, které obsahuje daná lokalita a na které je možno zprostředkovat konzolový přístup. Vzhledem k tomu, že každá lokalita může obsahovat rozlišné prvky, seznam zařízení je načítán z konfiguračního souboru *cons-devices.conf.* Ke každému zařízení je také uveden způsob, kterým se na síťový prvek dostane (pomocí sériové linky nebo pomocí TCP/IP protokolu).

Konfigurace pomocí souboru *cons-devices.conf* je poměrně snadná. Každý síťový prvek se uvádí na jednom řádku, kdy první argument (oddělen jednou mezerou) je název zařízení. Dále následuje způsob, jak se zařízením komunikovat. Způsoby jsou zpravidla dva. Pomocí TCP/IP protokolu, v tom případě se do konfiguračního souboru zapíše klíčové slovo *telnet,* nebo pomocí sériové linky, kdy se zapisuje klíčové slovo *serial.* Dále následuje další parametr oddělen jednou mezerou, který označuje v případě TCP spojení IP adresu a port a v případě sériové linky cestu k ovladači sériového portu[6].

Příklad konfiguračního souboru pro dvě zařízení.

```
pc1 telnet 10.0.0.100:10001
r1 serial /dev/ttyS0
```

6.2 Komunikační protokol

Jedná se o textově orientovaný protokol využívající TCP/IP linku. Před propojením se síťovým prvkem je nutno zaslat autentizační parametry, které se skládají ze čtyř částí oddělených znakem konce řádku. První parametr vyznačuje název zařízení, které požadujeme propojit. Další parametr reprezentuje session ID, přes které je uživatel připojen ke svému účtu na webovém serveru. Následující parametr je tutor mód, který vyznačuje v jakém režimu má konzole pracovat (off,observer,exclusive,shared) **.** Čtvrtý parametr obsahuje ID rezervace uživatele.

[6] Více detailů o architektuře Konzolového serveru lze naleznout v Diplomové práci Radka Nováka[2]

Příklad autentizace oproti konzolovému serveru:

```
r5@vsb
5a89feb3983bd8b38ce89a153075
off
13@vsb
```

Po úspěšné autentizaci se uživatel ocitne ve virtuální konzoli, kdy se každý stisknutý znak odešle na vzdálené zařízení. Práce na zařízení končí odpojením se z konzolového serveru.

6.3 Návrh scénáře testování konzolového serveru

Rozhodl jsem se, že vytvořím záznam o fiktivním vzdáleném počítači, který přidám do seznamu zařízení. Po autentizaci bude konzolový server vést datový tok zpět do testovací aplikace, která má za úkol porovnat zaslaná data s přijatými daty. Pokud přijatá data budou odpovídat zaslaným, funkčnost konzolového serveru je v pořádku. Zařízení pojmenuji např. *test@device*. Několik problémů však skýtá samotná autentizace. Konzolový server požádá o autentizaci PHP skript *verify.php*, který ověří správnost session ID uživatele, jež chce na zařízení pracovat a stejně tak ID rezervace. PHP skript konzolovému serveru po té vrací odpověď ve tvaru „1“ nebo „0“, zda-li se autentizace povedla, či nikoliv. Jelikož, se od aplikace neočekává jednání jako od uživatele (nutnost založení rezervace apod.), je nutno v PHP skriptu udělit výjimku pro přesně dané session ID, které vrátí vždy úspěšnou autentizaci.

Záznam o fiktivním zařízení může mít tvar.

```
test@device telnet 157.0.0.52:10001
```

Fragment kódu výjimky nutno dodat na začátek PHP skriptu.

```
if($sessionID.equals($tajneSID)){
      print(„1“);
      exit;
}
```

6.3.1 Problém s autentizací

Při počáteční analýze, kdy jsem s konzolovým serverem komunikoval prostřednictvím programu *telnet*, jsem si neuvědomil, že pracuji na lokálním počítači. Při první implementaci testovacího zařízení, jsem zjistil, že konzolový server se nechová tak, jako bych očekával (vracel neúspěšnou autentizaci). Po hlubší analýze jsem zjistil, že konzolový server autentizuje pouze takové TCP spojení, které vyšlo se stejného počítače na kterém služba běží (*localhost*). Pro tento případ jsem navrhnul a implementoval aplikaci, která naslouchá na lokálním počítači a pomoci jednoduchého protokolu vytvoří proxy rozhraní, jehož prostřednictvím lze komunikovat s konzolovým serverem jako z lokálního počítače.

6.4 Implementace směrovacího zařízení

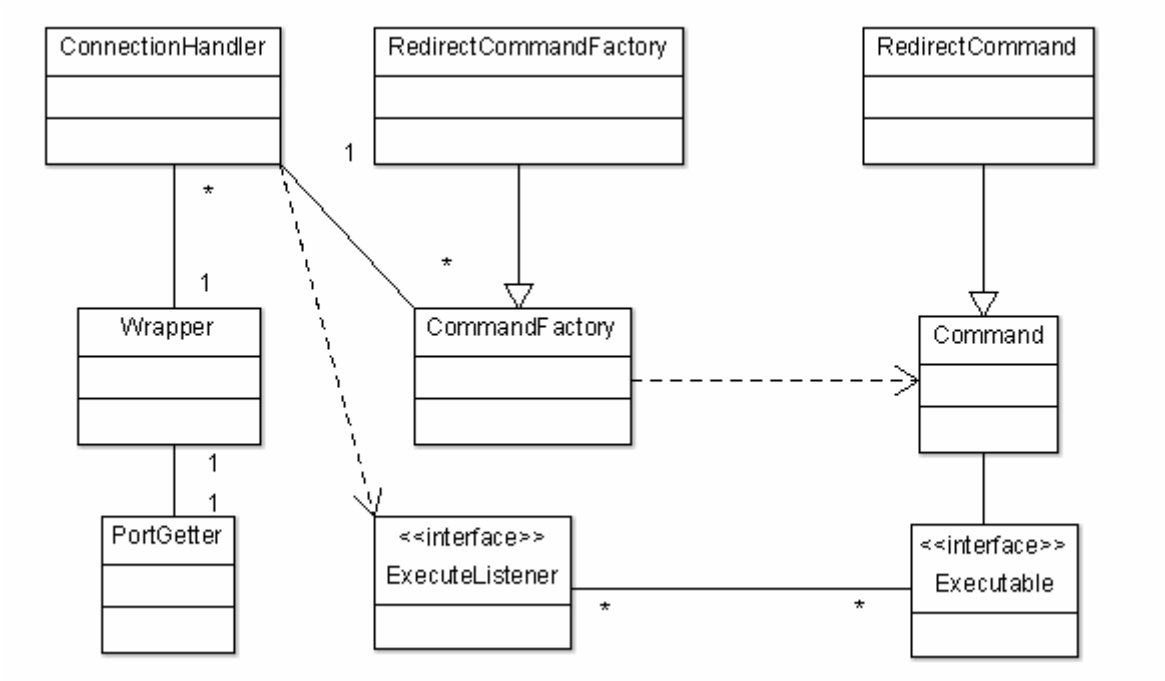
Jak už bylo zmíněno v předchozí podkapitole (6.3.1) z důvodu problému s autentizací oproti konzolovému serveru jsem byl nucen implementovat směrovací zařízení, jež naslouchá na některém z TCP portu lokality. Testovací aplikace bude mít za úkol připojit se k směrovacímu zařízení a zaslat požadavek o přesměrování na jinou TCP adresu. Směrovací zařízení požadavek přijme a pokusí se navázat požadované spojení. Pokud je spojení navázáno, zařízení vytvoří novou spojovací linku reprezentovanou stávajícím připojením ke konzolovému serveru a vytvoření novým serverovým socketem. Po té zašle kladnou odpověď s číslem portu, na které se má testovací aplikace připojit. Pokud spojení nebylo navázáno, zařízení odešle testovací aplikaci zprávu o nedostupnosti požadované adresy.

Ukázka komunikace se směrovacím zařízením:

```
// testovací aplikace
REDIRECT ADDRESS=localhost PORT=10000

// odpověď ze směrovacího zařízení
200 - Connection established port=5001
```

6.4.1 Architektura

Architekturu aplikace znázorňuje zjednodušený UML třídní diagram na obrázku 8.



Obrázek 8 – Architektura směrovací aplikace

6.4.2 Význam důležitých tříd

Třída Wrapper:

Jedná se o výchozí třídu, která má za úkol přijímat příchozí spojení a rozdělovat je pomocí třídy *ConnectionHandler* do samostatných vláken.

Třída ConnectionHandler:

Třída se stará o obsluhu klienta jako takového. Obsahuje seznam příkazů, který je reprezentován návrhovým vzorem továrna. Továrnu jsem zvolil z důvodu možného rozšíření aplikace. Pokud budeme chtít aplikaci rozšířit o nový příkaz, stačí implementovat novou továrnu dědící z třídy *CommandFactory* a zaregistrovat jí v hash tabulce třídy *ConnectionHandler*. V momentální situaci systém obsahuje pouze příkaz *REDIRECT*, reprezentován třídami *RedirectCommandFactory* a *RedirectCommand*.

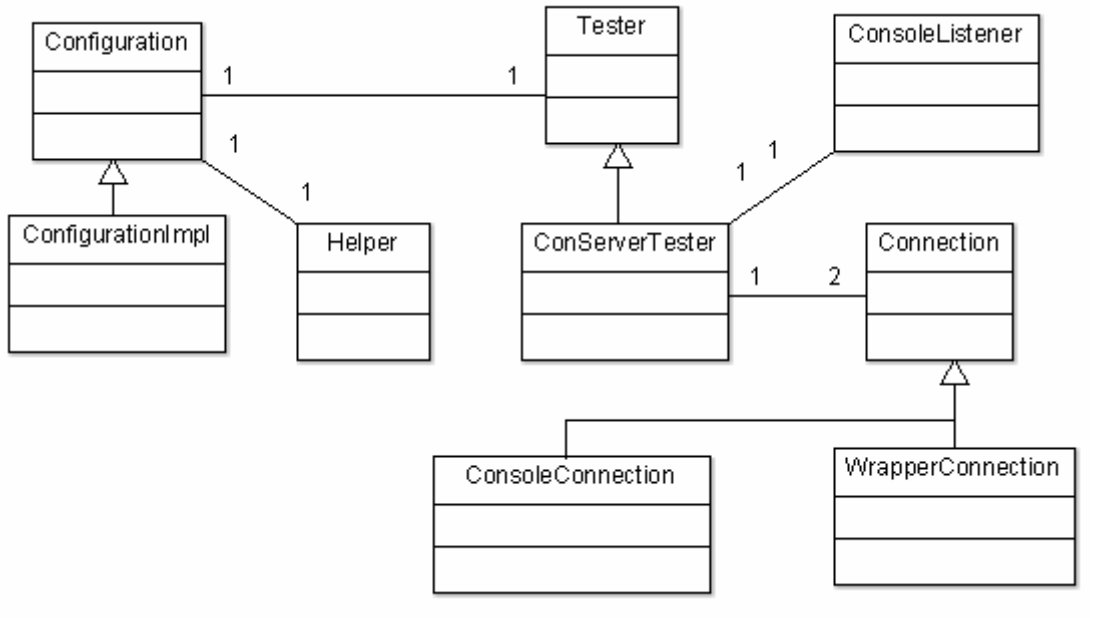
Třída PortGetter:

Jde o třídu, jež má za úkol přidělovat neobsazené TCP porty. Jelikož se jedná o aplikaci, která využívá více vláknový přístup, při obsluze několika klientů najednou se může stát, že dva klienti využívají stejný TCP port, což by mohlo způsobit možný pád aplikace. V konfiguračním souboru je definován rozsah přidělených portů aplikaci. Pokud klient zažádá o přidělení portu a v momentální situaci není žádný port k dispozici, musí počkat až jiný klient uvolní některý s portů.

6.5 Implementace testovací aplikace

6.5.1 Architektura

Architekturu implementace testovacího skriptu lze shlédnout na UML diagramu tříd v obrázku 10 uvedeného níže.



Obrázek 9 – Architektura testovací aplikace konzolového serveru

6.5.2 Význam důležitých tříd[7]

Třída: Configuration

Jedná se o abstraktní třídu, která definuje objekt konfigurace aplikace. Třída poskytuje svou implementaci pomocí vnitřní třídy *ConfigurationImpl*.

Třída: ConfigurationImpl

Jedná se o konkrétní implementaci konfigurace aplikace, tedy dědí z třídy Configuration. Pro konfiguraci jsem zvolil XML dokument.

Třída: Connection

Tato třída definuje základní operace nad TCP spojením (*connect*, *close* apod.). Třídu využívají podtřídy, *ConsoleConnection* a *WrapperConnection*, z důvodu zbytečného kopírování identického kódu.

Třída: ConsoleConnection

Třída reprezentuje TCP spojení mezi aplikací a Konzolovým serverem a zároveň je potomkem třídy *Connection*. Poskytuje zasílání autentizace oproti Konzolovému serveru a libovolné zprávy.

Třída: WrapperConnection

Jde o třídu, která definuje spojení mezi aplikací a směrovacím zařízení. Třída poskytuje vytvoření proxy rozhraní na úrovni směrovacího zařízení, kdy po požadavku o přesměrování vrací instanci třídy *ConsoleConnection*.

Třída: ConsoleListener

Jedná se o třídu, která reprezentuje TCP naslouchání pro příchozí spojení od Konzolového serveru. Třída je tvořena serverovým socketem, přes který *bufferuje* přijímána data a po odpojení je poskytuje řídící třídě na porovnání.

Třída: ConServerTester

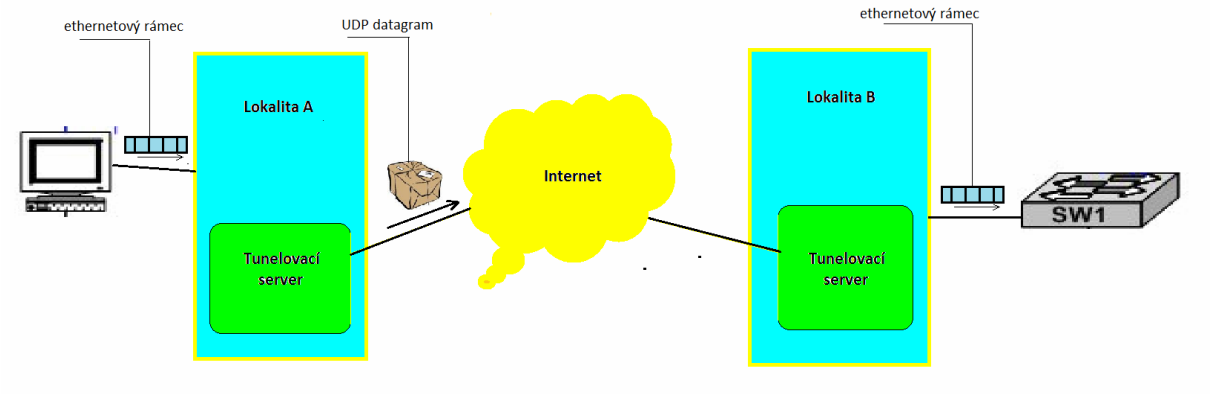
Jedná se o třídu, která řídí celý průběh testu. Zajišťuje správné formátování výstupních dat, komunikaci s komponentami, kontrolu odpovědí, zachycení a správnou reakci na možný výskyt výjimek, načítání konfigurace apod.

[7] Neuvedené třídy mají obdobný význam jako stejnojmenné třídy uvedeny v předchozích kapitolách (viz. kapitoly 4.5.4, 5.4.4, 6.5.2)

7 Tunelovací server

Vzhledem k tomu, že Virtlab je distribuovaný systém, vznikla potřeba přenosu směrování datového toku mezi lokalitami. Pro příklad si představme, že máme v topologii počítač, který je v lokalitě A přepínač, který je v lokalitě B. Tyto dvě zařízení jsou v topologii propojeny mezi sebou, a však v realitě jsou tyto zařízení desítky kilometrů od sebe. Tunelovací server (respektive tunelovací servery), který je umístěn v obou lokalitách má za úkol vytvořit virtuální tunel mezi lokalitami, který tyto dvě zařízení skrze internet propojí tak, že ve skutečnosti vypadá jako by síťové prvky byly navzájem fyzicky propojeny. Tento tunel je vytvářen nad technologií VLAN a přenos je realizován pomocí principu Q-in-Q, což znamená, že pokud zařízení z lokality vyšle ethernetový rámec, tunelovací server tento rámec přijme, a pokud rámec nenáleží do lokality A, zabalí ho do UDP datagramu, a pošle přes internet tunelovacímu serveru v lokalitě B. Tunelovací server v lokalitě B má za úkol tento zaslaný datagram přijmout, rozbalit a poslat svému zařízení opět ve formě ethernetového rámce[8] (viz. Obrázek 10).

Jelikož se jedná o manipulaci na nižších síťových vrstvách, tunelovací server je implementován v jazycích C/C++.



Obrázek 10 – Princip činnosti tunelovacího serveru

7.1 Architektura a konfigurace

Tunelovací server lze rozdělit do dvou komponent, a to démona běžícího na pozadí serveru a konzole přes kterou se démon konfiguruje. Démon slouží k samotnému směrování dat, naslouchá na přednastaveném síťovém rozhraní a přeposílá data jinému tunelovacímu serveru, nebo sám sobě, pokud se jedná o směrování v rámci stejné lokality. Konzole slouží ke konfiguraci démona, nastavují se zde rozhraní na kterých má démon naslouchat, kam data přeposílat apod. Konzole je implementována ve více vláknové technologii, tedy pro každého uživatele se vytvoří samostatné vlákno.

Tunelovací server obsahuje škálu konfiguračních souborů. Nás však zajímají pouze dva z nich, a to *localvlans.conf*, kde se nastavují zařízení, síťová rozhraní a k nim přiřazená VLAN ID dané lokality a soubor *trunkport.conf,* kterým se definuje síťové rozhraní, jež umožňuje přístup k internetu.

[8] Více informací o funkcionalitě tunelovacího serveru lze naléznout v bakalářské práci Václava Bortlíka[3]

V souboru *localvlans.conf* je každé zařízení zapsáno na samostatný řádek, kde prvním argumentem je úplný název zařízení (např. zarizeni1@vsb) a znakem „:“ odděleno síťové rozhraní. Jako druhý argument se zde nachází VLAN ID, které je odděleno jednou mezerou. Soubor *trunkport.conf* obsahuje pouze řetězec, který odpovídá názvu síťového rozhraní, jež umožňuje přístup k internetu (např. eth0).

Příklad konfigurace pomocí souboru *localvlans.conf:*

```
pc1@lokalita:eth0.1000 1000
r1@jinalokalita:eth0 1500
```

7.2 Komunikační protokol

Komunikovat s tunelovacím serverem lze pouze přes přístupovou konzoli, která běží nad TCP/IP protokolem a standardně naslouchá na portu 40001. Konzole slouží převážně pro nastavení přesměrování datového toku mezi lokalitami (viz. kapitola 7.1), a však lze v ní také dynamicky načítat konfigurační soubory, rušit směrování, či vypisovat právě běžící směrování. Jedná se v podstatě o textově orientovaný protokol, kdy každou volbu oddělujeme (i s argumenty) znakem ukončení řádku <LF>. Všechny volby a jejich argumenty jsou vypsány v seznamu níže.

- **redir <interface> <interface>** - příkazem definujeme přesměrování ze síťového rozhraní na jiné síťové rozhraní, jako argument předáváme celý řetězec jednoho řádku z konfiguračního souboru *localvlans.conf* (bez VLAN ID)
- **noredir <interface>** - příkazem se ruší stávající přesměrování, jako argument předáváme opět řetězec s konfiguračního souboru
- **reload <conf>** – příkazem se za běhu znova načte konfigurační soubor (vlans,port_setter,serial>
- **show <option>** - na konzoli vypíše uvedenou volbu
- **help [option]** – na konzoli vypíše všechny volby, nebo informace o argumentech dané volby
- **log,showlog,dellog,nolog** – volby slouží k práci s logováním
- **exit** – slouží ke korektnímu odpojení z konzole

7.3 Návrh scénáře testování

Vzhledem k nepřípustnosti démona tunelovacího serveru lze jen velmi obtížně ověřit správnou funkčnost služby. Jediná cesta k ověření spočívá v nasimulování funkčnosti na předem vytvořených síťových rozhraní. Tedy vytvoříme dvě síťová rozhraní, které následně přesměrujeme s konzole tunelovacícho serveru a pokud budeme na jednom z rozraní zasílat ICMP zprávy, na druhém rozhraní by se měl objevit alespoň ARP packet. Tento paket odchytíme pomocí programu TCPDUMP, který slouží právě pro tyto účely. ICMP zprávy můžeme snadno generovat programem PING.

Jelikož se jedná o interní záležitost, navrhnul jsem rozšířit směrovací zařízení z kapitoly 6.4, právě o podporu spouštění programů PING a TCPDUMP. Testovací aplikace tedy bude mít za úkol:

- připojit se ke konzoli tunelovacího serveru a přesměrovat předem vytvořené síťové rozhraní
- vytvořit spojení se směrovací aplikací a spustit program PING pro vhodný počet packet
- vytvořit nové připojení ke směrovací aplikaci, spustit program TCPDUMP a po určitý čas odchytávat přijaté packety
- porovnat, zda-li v přijatých datech není alespoň jeden packet ARP z vhodnou cílovou adresou
- opět se připojit ke konzoli tunelovacího serveru a zrušit přesměrování

K vytvoření VLAN síťových rozhraní využijeme příkazu *vconfig* (viz. [8]), přiřadíme jim IP adresy a uvedeme je v konfiguračním souboru tunelovacího serveru (*localvlans.conf*) .

Příklad konfigurace VLAN rozhraní:

```
vconfig add eth0 1000
ip address add 10.0.0.1/24 brd + dev eth0
ip link set eth0.1000 up
vconfig add eth0 1500
ip address add 20.0.0.1/24 brd + dev eth0
ip link set eth0.1500 up
```

V konfiguračním souboru *localvlans.conf:*

```
testpc1@vsb:eth0.1000 1000
testpc2@vsb:eth0.1500 1500
```

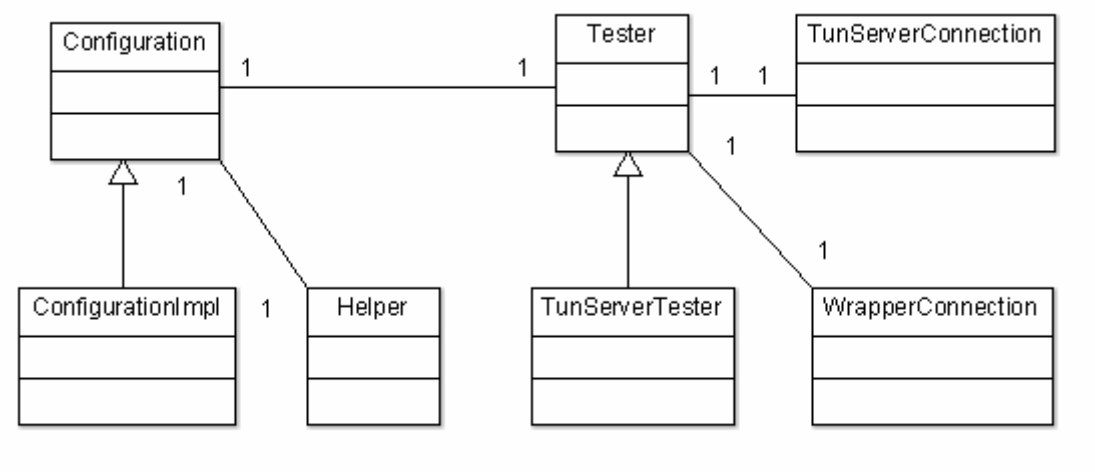
7.4 Implementace testovací aplikace

7.4.1 Rozšíření směrovací aplikace

Jak už jsem zmínil v předchozí podkapitole (7.3) o rozšíření směrovací aplikace o podporu spouštění příkazu PING a TCPDUMP. Rozšíření proběhlo vyděděním z třídy *Command* a *CommandFactory*. Třídy jsem pojmenoval *PingCommand*, *PingCommandFactory*, *TcpDumpCommand* a *TcpDumpCommandFactory* viz. kapitola 6.4. Proběhla také registrace továren pod názvy PING a TCPDUMP, což znamená, že příkazy jsou dostupné právě pod těmito názvy v komunikačním protokolu.

7.4.2 Architektura testovací aplikace

Strukturu testovací aplikace popisuje zjednodušený UML diagram na obrázku níže (Obrázek 11).



Obrázek 11 – Architektura testovací aplikace tunelovacího serveru

7.4.3 Význam důležitých tříd

Třída TunServerConnection:

Jedná se o třídu, která slouží k manipulaci s konzolí tunelovacího serveru. Pro navázání TCP spojení je využito třídy *java.net.Socket*. Obsahuje zejména metody pro vytvoření, či zrušení přesměrování síťových rozraní.

Třída WrapperConnection:

Jde o třídu, která reprezentuje TCP spojení se směrovacím zařízením. Obsahuje především metody k spouštění příkazů PING a TCPDUMP. Metoda pro spouštění programu TCPDUMP zároveň vrací instanci třídy, která implementuje rozhraní Connection. Rozhraní Connection definuje obecné TCP spojení, je tedy požito pro stahování dat z výstupu programu *tcpdump*.

Ostatní stejnojmenné třídy mají obdobný význam jako z předchozích implementací testovacích aplikací. Řešení detailů, jako jsou maximální časová prodleva (timeout), formátování výstupů, či výpočty časových prodlev, jsou pro všechny testovací aplikace naprosto identické. Třídy pro manipulaci s těmito věcmi jsou obsaženy v balících *util*.

8 Výstupy a spouštění testovacích aplikací

Jako primární výstup všech aplikací jsou textové informační zprávy o průběhu, či chování testovaných služeb. Tyto zprávy však nelze rozumným způsobem zpracovat výpočetní technikou a tak globálně diagnostikovat chování služby v průběhu času. Proto jsem zavedl do každé s aplikace datový výstup, který bude možno zpracovat výpočetní technikou a v průběhu času z něj bude možno případně vykreslit graf.

Každá aplikace monitoruje specifickou službu a proto není jednoduché sjednotit jejich datový výstup. Jedna veličina však spojuje všechny služby a tou je čas. Proto jako hlavní datový výstup všech aplikací je čas provedení testu. Ostatní výstupy jsou specifické pro danou službu. U mazacího serveru sledujeme poměr identičnosti vyslaných znaků oproti znakům přijatým, u rezervačního serveru čas vykonání jednotlivých dotazů (z těchto dat lze odvozovat vytížení služby) , u konzolového serveru opět poměr zaslaných dat oproti datům přijatým a u tunelovacího serveru počet přijatých packet ARP.

Ze všech implementovaných testovacích aplikací byl vytvořen balíček v podobě *jar* souboru. Spouštějí se tedy standardně pomocí příkazu *java*. K aplikacím náleží také argumenty spouštění, které jsou implicitně načítány z konfiguračního souboru dané aplikace, ale lze je explicitně přepsat zadáním argumentu při spouštění (viz. *readMe.txt* přiložené ke každé aplikaci, příp. parametr –h).

Příklad spouštění testovací aplikace pro mazací server:

```
java –jar EraseServerTester.jar –H localhost –p 50001
```

9 Závěr

Všechny výše zmíněné testovací aplikace byly testovány na virtualizované lokalitě systému Virtlab. Testování zahrnovalo vyzkoušení nejdůležitějších funkcí i při ztížených podmínkách (uměle vyvolána nefunkčnost komponent, simulován výpadek v síti apod.). Při tomto testování bylo zjištěno pár drobných chyb a nedostatků, které byly následně opraveny. Aplikace budou nasazeny do provozu již v akademickém roce 2011/2012 Martinem Milatou.

Hardwarové zařízení ASSSK bylo v průběhu vypracovávání této bakalářské práce vyloučeno ze systému Virtlab, a proto byla po konzultaci s vedoucím bakalářské práce problematika tohoto zařízení vyloučena i z této práce. Na závěr bych chtěl upozornit na nedostatečnou, či nepřesnou dokumentaci, co se týče zejména komponent virtuální laboratoře.

10 Reference

[1] ECKEL, Bruce. *Thinking in Java : 3rd Edition* [online]. Upper Saddle River, New Jearsey 07458 : Pesident, MindView Inc., 2003 [cit. 2011-04-16]. Dostupné z WWW: <http://ii.iinfo.cz/r/k/TIJ3.pdf>. ISBN 0-13-100287-2.

[2] NOVÁK, Radek. *Robustní reimplementace prototypového řešení serverových komponent systému Virtlab*. FEI VŠB-TU Ostrava, 2010. 71 s. Diplomová práce. Vysoká Škola Báňská. Dostupné z WWW: <http://infra2.cs.vsb.cz/vl-wiki/images/2/26/Novak-DP-2010.pdf>.

[3] BORTLÍK, Václav. *Distribuované spojovací pole pro virtuální laboratoř počítačových sítí*. FEI VŠB-TU Ostrava, 2008. 29 s. Bakalářská práce. Vysoká Škola Báňská. Dostupné z WWW: <http://infra2.cs.vsb.cz/vl-wiki/images/9/92/Bortlik-diplomka.pdf>.

[4] *VirtlabWiki* [online]. 2007 [cit. 2011-04-16]. Dostupné z WWW: <http://infra2.cs.vsb.cz/vl-wiki/index.php>.

[5] BENNETT, Derrick, et al. *Nagios 3 Enterprise Network Monitoring*: Including Plug-Ins and Hardware Devices. USA : SYNGRESS, 2008. 348 s. ISBN 978-1-59749-267-6.

[6] TURNBULL, James. *Pro Nagios 2.0*. USA : APress, 2006. 400 s. ISBN 978-1-59059-609-8.

[7] SCHMULLER, Joseph. *Myslíme v jazyku UML*. 1. vyd. Praha : Grada, 2001. 359 s. ISBN 80-247-0029-8.

[8] *AbcLinuxu* [online]. 1999 [cit. 2011-04-16]. Dostupné z WWW: <http://www.abclinuxu.cz/>.

11 Příloha – Vstupní argumenty aplikací

11.1 Mazací server

- -v -version - vypíše na konzoli verzi a informace o aplikace
- -h -help - vypíše na nápovědu pro spouštění skriptu
- -H <IP or DNS> -host - IP adresa lokality kde se nachází mazací server
- -p <number> -port - TCP port mazacího portu (standardně 60002)
- -V -verbose - upovídaný výstup
- -t <number> -timeout - nastavení timeout

11.2 Rezervační server

- -v -version - vypíše na konzoli verzi a informace o aplikace
- -h -help - vypíše na nápovědu pro spouštění skriptu
- -H <IP or DNS> -host - IP adresa lokality kde se nachází rezervační server
- -l <name> -locality - název lokality
- -p <number> -port - TCP port rezervačního serveru (standardně 40001)
- -V -verbose - upovídaný výstup
- -t <number> -timeout - nastavení timeoutu

11.3 Konzolový server

- -v -version - vypíše na konzoli verzi a informace o aplikace
- -h -help - vypíše na nápovědu pro spouštění skriptu
- -H <IP or DNS> -host - IP adresa lokality kde se nachází konzolový server
- -p <number> -port - TCP port konzolového serveru (standardně 10000)
- -lp <number> -listened port - port na kterém má aplikace naslouchat
- -V -verbose - upovídaný výstup
- -t <number> -timeout - nastavení timeoutu

11.4 Tunelovací server

- -v -version - vypíše na konzoli verzi a informace o aplikace
- -h -help - vypíše na nápovědu pro spouštění skriptu
- -V -verbose - upovídaný výstup
- -t <number> -timeout - nastavení timeoutu
- -int1 <interface> -interface1 - síť. první rozhraní k přesměrování
- -int2 <interface> -interface2 - síť. druhé rozhraní k přesměrování
- -pa <IP address> -ping address - IP adresa na kterou se mají generovat ICMP zprávy
- -H <IP or DNS> -host - IP adresa lokality kde se nachází tunelovací server
- -p <number> -port - TCP port tunelovacího serveru (standardně 50001)